INA: Assignment #1

Nejc Ločičnik, 63180183

University of Ljubljana — March 29, 2024

1 Connected Components

First the mathematical induction over c for the formula $n - c <= m$:

- Base step: $c = 1$ The formula holds for $c = 1$, because a fully connected graph with n nodes and a minimum amount of edges is a tree, in which case $n - 1 = m$ (equality holds). If the graph is not a tree, meaning we start with a tree, but add some edges between existing nodes, then $n - 1 <= m$ holds (< as we are adding edges). The maximum amount of edges we can add to a simple undirected graph is $\binom{n}{2}$, which represents a fully connected graph, where every node is directly connected to every other node.

- Assumption this holds for an arbitrary c: $c = k$ -> $n - k <= m$

- Inductive step: $c = k+1$ In this case we have a graph G with $k+1$ connected components, meaning we can represent each component (subgraph) as $G_1, G_2, ...G_{k+1}$, where each component has $n_1, n_2, ...n_{k+1}$ nodes and $m_1, m_2, ...m_{k+1}$ edges.

We know from the base step that the formula holds for $c = 1$, so we can rewrite the formula as:

$$\sum_{i=1}^{k+1} (n_i - 1) <= \sum_{i=1}^{k+1} m_i$$

which we can rewrite as:

$$\sum_{i=1}^{k+1} n_i - (k+1) <= \sum_{i=1}^{k+1} m_i$$

$$n - (k+1) <= m$$

$$n - c <= m$$

As for the second inequality: $m <= \binom{n-c+1}{2}$, we check the upper and lower bound: a fully connected and disconnected graph. In a fully connected graph, we have $c = 1$ and $m = \binom{n}{2}$. If we start removing edges $m <= \binom{n}{2}$ will hold.

In a fully disconnected graph we have $m = 0$ and $n = c$ (each node is an isolated component, assuming no loop connections), then:

$$m <= \binom{n-c+1}{2} = \binom{1}{2} = 0 = m$$

Criterion:

1. $n - 1 = m$ is the minimum amount of edges needed to be fully connected (tree), very idealistic if we are talking about randomly adding edges

2. the minimum amount of edges needed to **ensure** a fully connected graph (when randomly adding edges, while being restricted to a simple graph) is having a single edge (bridge) between 2 fully connected components (we can't add anymore edges to the fully connected components so we are forced to connect them).

The criterion doesn't seem all that practical, most real-world networks have way too many nodes and edges and their structure is dictated by other information/context not seen through by the number of nodes/edges/components.

2 Node linking probability

Since the edges are placed independently between each pair of nodes i and j with the probability p_{ij}, we can use the definition for expected value to express the expected node degree $\langle k_i \rangle$ as a sum of probabilities:

$$\langle k_i \rangle = \sum_{j \neq i} p_{ij}$$

Since p_{ij} is proportional to $v_i v_j$, we can replace p_{ij} with $C v_i v_j$ for some constant C:

$$\langle k_i \rangle = \sum_{j \neq i} C v_i v_j$$

We can then factor out the v_i, since it doesn't depend on j:

$$\langle k_i \rangle = v_i \sum_{j \neq i} C v_j$$

$\sum_{j \neq i} C v_j$ is a constant in regards to i, so we can denote it as another constant C' (different notation, because the constants can be different).

$$\langle k_i \rangle = C' v_i$$

Hence the node degree $\langle k_i \rangle$ is proportional to v_i.
Using what we've shown in *a)*, we can express v_i in terms of $\langle k_i \rangle$, with some constant C:

$$v_i = C \langle k_i \rangle$$

which we can insert into the expression for p_{ij}:

$$p_{ij} \propto v_i v_j = (C_i \langle k_i \rangle)(C_j \langle k_j \rangle) = C_i C_j \langle k_i \rangle \langle k_j \rangle = C' \langle k_i \rangle \langle k_j \rangle$$

$C_i C_j$ is a constant so we can replace them with C', which shows that p_{ij} is proportional to the product of the expected node degrees ($\langle k_i \rangle \langle k_j \rangle$) of nodes i and j.

The derived result demonstrates a fundamental relationship between the probability of edge formation and the expected degrees of the nodes in the graph. It implies that nodes with higher expected degrees are more likely to attract edges in the network. This tendency can lead to the over representation of high-degree nodes in the graph (which as an example of real-world network analysis, results in the friendship paradox).

3 Random node selection

The used network representation is adjacency list, which is best for efficient algorithm implementations.

```
def random_node(G): # G is represented with adjacency list
    node_idx = random.randint(0, len(G)-1)
    neighbour_idx = random.randint(0, len(G[node_idx])-1)
    return G[node_idx][neighbour_idx]
random_node(G)
```

This algorithm should work for all types of graphs, for directed graphs we would need to add another randint which would choose either the predecessors or the successors list. Multi-graphs should also be covered as they simply add repeated nodes to the lists in case of same edges. Loops should also be fine as we add the node twice to the list (so we keep the $2m$ in the needed probability $\frac{k_i}{2m}$).

It also runs in constant time, since we only pick random integers within the limits (amount of nodes and degree per node) and then simply return the selected node by indexing.

4 Weak & Strong connectivity

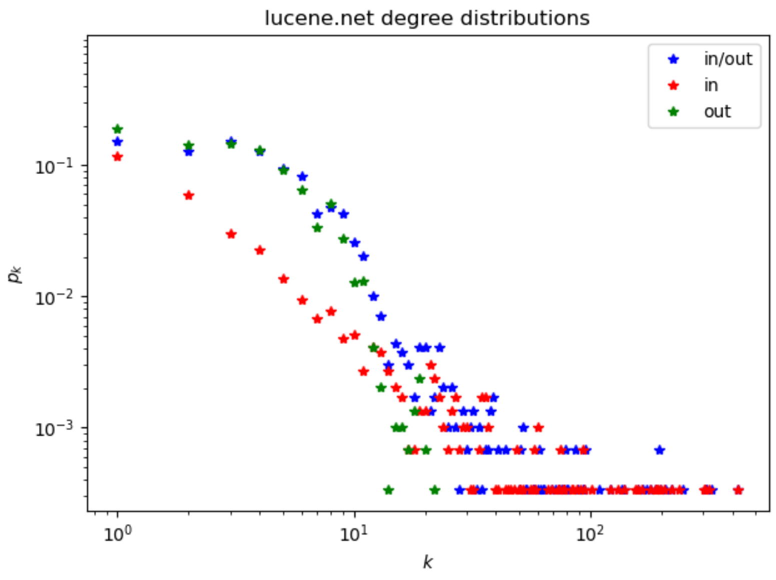
1. Following links in any direction in a directed graph would result in a weakly connected component (same if we'd look for connected components in an undirected graph).
2. Following links only in the right direction in a directed graph would result in DFS tree rooted at the starting node. The tree will include all nodes reachable from the starting node. (NOT a strongly connected component).
3. Following links only in the opposite direction in a directed graph would result in a list of all nodes that can reach the starting node when moving along (in the right direction) the edge. (again NOT a strongly connected component).

Python code prinout:

```
def DFS(G, i, visited, direction, stack):
    visited.add(i)
    for neighbour in G[i][direction]:
        if neighbour not in visited:
            DFS(G, neighbour, visited, direction, stack)
    stack.append(i)

def find_scc(G):
    visited = set()
    stack = []
    for node in range(len(G)): # visit every node
        if node not in visited: # if it wasn't visited yet
            DFS(G, node, visited, 0, stack)
        #print(len(stack), len(visited))
    visited.clear()
    sccs = []
    #print(stack)

    while stack:
        node = stack.pop()
        if node not in visited:
            #print(visited)
            scc_stack = []
            DFS(G, node, visited, 1, scc_stack)
            sccs.append(scc_stack)
            visited.update(scc_stack)

    return sccs

G = read_graph('enron.net', 'directed')
sccs = find_scc(G)

len(sccs), len(max(sccs, key=lambda lst: len(lst)))
```

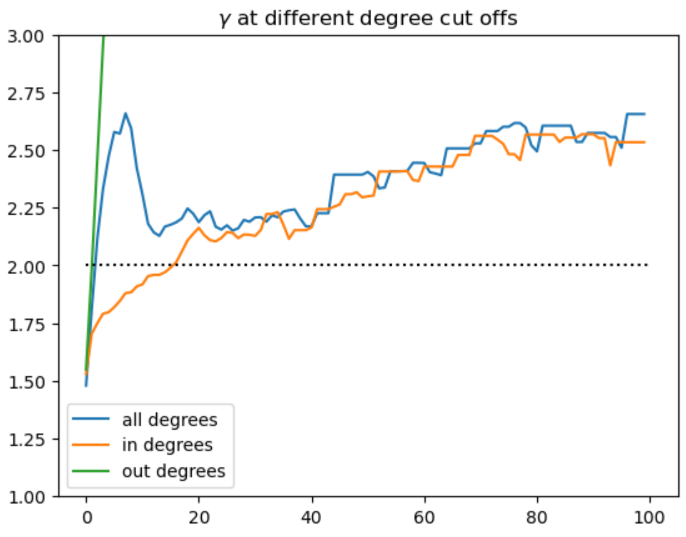
My implementation and the networkx implementation find 78058 strongly connected components, where the largest one contains 9164 nodes. The size of the largest strongly connected component seems a bit absurd, it is probably the result of forwarding e-mails and kind off shows the scale of the hierarchical structure in the company. The sheer amount of strongly connected components seems reasonable, coworkers probably send e-mails to everyone they work with.

5 Is Java software scale-free?



1. The in-degree distribution seems to be scale-free as it follows the power-law distribution (straight line, wide tail, doesn't fall off at the top like the other two). This probably happens, because we have central classes (hubs) on which a lot of other classes depend on (so a few nodes with high in-degree and a lot of nodes with lower in-degree)
2. The out-degree distribution doesn't seem to follow the power-law distribution so we can assume it's not scale free. This is probably because classes in general don't depend on too many other classes for their functionality (just good code practice), so the out-degrees would be more similar to a random graph and would have much lower max degree.
3. Total degree distribution doesn't seem to follow the power-law distribution, so we again assume it's not scale free. This is probably because we combine both in and out degree together, meaning the out-degree distribution will probably disturb the scale-free nature of the in-degree distribution.

The calculated γ. for the in-degree (at $k_{min} = 5$) is 1.7976323613898262.



The code used to plot distribution and calculate gamma.

```python
def plot(G, ax, c, type=0):
    nk = {}
    if type == 0: # in/out degree
        for _, k in G.degree():
            if k not in nk:
                nk[k] = 0
            nk[k] += 1
    elif type == 1:
        for _, k in G.in_degree(): # change this to only in degrees
            if k not in nk:
                nk[k] = 0
            nk[k] += 1
    elif type == 2:
        for _, k in G.out_degree(): # change this to only out degrees
            if k not in nk:
                nk[k] = 0
            nk[k] += 1

    ks = sorted(nk.keys())

    ax.loglog(ks, [nk[k] / len(G) for k in ks], '*', color=c)
    ax.set_xlabel('$k$')

G = nx.DiGraph(nx.read_pajek('lucene.net'), name="lucene.net")
fig, ax = plt.subplots(1, 1, tight_layout=True)
plot(G, ax, 'b', 0)
plot(G, ax, 'r', 1)
plot(G, ax, 'g', 2)
ax.set_title("lucene.net␣degree␣distributions")
ax.set_ylabel('$p_k$')
ax.legend(['in/out', 'in', 'out'])
plt.show()

def calc_gamma(G: nx.DiGraph, cut_off: int, type=0):
    if type == 1:
        degrees = [k for _, k in G.in_degree() if k > cut_off]
    elif type == 2:
        degrees = [k for _, k in G.out_degree() if k > cut_off]
    else:
        degrees = [k for _, k in G.degree() if k > cut_off]
    if degrees:
        k_min = min(degrees)
        n_dot = len(degrees)
    else:
        return float('inf')
    #print(k_min)
    return 1+n_dot*sum([log(ki/(k_min - 1/2))*(ki >= k_min) for ki in degrees])**(-1)

print("in/out␣degree:␣", calc_gamma(G, 4))
print("in␣degree:␣", calc_gamma(G, 4, 1))
print("out␣degree:␣", calc_gamma(G, 4, 2))
```

6 Five networks problem

The first graph to find is the realization of the ER random graph, because we can simply compare the max degree with the other graphs, since the random graph degree distribution follows Poisson's distribution the max degree should be small and the next couple of degrees after it should have similar value. I identified it as network 2, with max degree of 22 and top 5 max degree of [22, 22, 21, 21, 20].

As we know 2 of the networks are directed, we can attempt to split the 4 networks into 2 separate groups by looking at some basic statistics. Networks 1 and 4 have a large amount of pendant nodes (45-50%), lower density and avg. degree compared to the other 2 networks, so I think they're the directed ones.

Of the other 2, network 5 clearly stands out, as it has an avg. degree of 132, highest density and almost no pendant nodes. I would argue this is IMDB Collaboration network, because I assume the actors represented in it, are mostly the well known ones and they tend to collaborate with each other a lot (so a well connected network). The other network is network 3, which has a clustering coefficient of 0, which is common for affiliation networks, so I think this is Flickr, a social affiliation network.

All that is left is the 2 directed graphs, network 1 and 4. Here I think network 1 is the citation network, because it has a higher avg. degree, density and clustering coefficient, which I think is more in line with a citation network. Authors citing one another and multiple citations per paper result in a more densely connected graph. The other reason is that network 4 has the lowest avg. degree, which I think is more in line with a web graph.
To sum up results:

1. Network - Sample of computer science citation network
2. Network - Realization of the ER random graph
3. Network - Flickr social affiliation network
4. Network - Small part of a university Web graph
5. Network - IMDb actors collaboration network

```
networks = ['network_1.adj', 'network_2.adj', 'network_3.adj', 'network_4.adj', 'network_5.adj']
for net_name in networks:
    with open(net_name, 'r') as file:
        file.readline()
        info = file.readline().strip().split("␣")
        n = int(info[1].replace(",", ""))
        m = int(info[4].replace(",", ""))
        file.readline()
        file.readline()

        type = 'edges'
        mm = 0
        if type == 'edges':
            G = [[] for _ in range(n)]
            for line in file:
                l = line.strip().split("␣")
                i = int(l[0])-1
                j = int(l[1])-1
                G[i].append(j) # from to - i is a predecessors of j
                G[j].append(i) # to from - j is a successor of i
                mm += 1

        degrees = [len(neigh) for neigh in G]

        print("Network␣name\t", net_name)
        print("Number␣of␣nodes\t", n)
        print("Number␣of␣edges\t", m)
        print("Avarage␣degree\t", (2*m)/n)
        print("Density\t\t", (2*m/(n*(n-1))))
        print("Max␣degree\t", max(degrees), sorted(degrees, reverse=True)[:5])
        print("Pendant␣nodes\t", degrees.count(1), degrees.count(1)/n)

        print("")

def read_graph(name):
    G = nx.Graph(nx.read_edgelist(name))
    #G = nx.convert_node_labels_to_integers(G, label_attribute="label")
    G.name = name
    return G

for net in networks:
    G = read_graph('./'+net)
    print(nx.average_clustering(nx.Graph(G)))
```

7 Who to vaccinate?

The second immunization scheme will provide better immunization, because by leveraging the structure of a social network, which has a property called The Friendship Paradox, where your friend is likely (on average) to have more friends that you (because high degree nodes are overrepresented). So by vaccinating a random acquaintance of the randomly selected individual, we are more likely to vaccinate a highly connected individual (individual with many friends), who has more influence on the spread of the disease.